EFI 1.1
EFI Byte Code Virtual Machine

Draft for Review

Version 0.7

April 10, 2001

Intel order number xxxxxx-001

Revision History

Revision	Revision History	Date
0.7	First public review. Clarified CALL instruction. Changed thunking implementation. Changed protocol.	4/10/01

Contributors	
Name	**Company**
Jay Bharadwaj	Intel, Compiler Group
Pasquale DeMaio	Microsoft
Mark Doran	Intel
Alice Kwong	Intel, Compiler Group
Derick Moore	LSI
Ravi Narayanaswamy	Intel, Compiler Group
Ken Reneris	Intel
Tom Rhodes	Compaq
Curtis E. Stevens	Phoenix Technologies
Roy Wade	LSI
Dong Wei	HP
Bryan Willman	Microsoft
Mike Kinney	Intel
Greg McGrath	Intel

Table of Contents

Tables

1 Introduction

1.1 Scope

This specification defines a Virtual Machine that can provide platform and CPU independent mechanisms for loading and executing EFI device drivers. When this specification is complete it will be integrated with the EFI 1.1 specification.

1.2 Related Information

The following publications and sources of information may be useful to you or are referred to by this specification:

- [REF-EFI] *EFI Specification*, Version 1.02, Intel Corporation, 2000, http://developer.intel.com/technology/efi.
- [REF-PCI] *PCI Driver Model,* Version 0.7, Intel Corporation, 2001
- [REF-PECOFF] *Microsoft Portable Executable and Common Object File Format Specification,* Revision 6.0, February 1999, by Microsoft Corporation.
- [REF-DEBUG] *EFI1.1 Debugger Support Specification,* Version 0.7, March 16, 2001.

1.3 Related Information

This document uses typographic and illustrative conventions described below.

1.3.1 Data Structure Descriptions

The Intel® Architecture processors of the IA-32 family are “little endian” machines. This means that the low-order byte of a multi-byte data item in memory is at the lowest address, while the high-order byte is at the highest address. Processors of the Itanium Processor Family (IPF) may be configured for both “little endian” and “big endian” operation. All implementations designed to conform to this specification will use “little endian” operation.

In some memory layout descriptions, certain fields are marked *reserved*. Software must initialize such fields to zero, and ignore them when read. On an update operation, software must preserve any reserved field.

1.3.2 Pseudo-Code Conventions

Pseudo-code is presented to describe algorithms in a more concise form. None of the algorithms in this document are intended to be compiled directly. The code is presented at a level corresponding to the surrounding text.

In describing variables, a *list* is an unordered collection of homogeneous objects. A *queue* is an ordered list of homogeneous objects. Unless otherwise noted, the ordering is assumed to be FIFO.

Pseudo-code is presented in a C-like format, using C conventions where appropriate. The coding style, particularly the indentation style, is used for readability and does not necessarily comply with an implementation of the *EFI Specification*.

1.3.3 Typographic Conventions

The following typographic conventions are used in this document to illustrate programming concepts:

`Prototype`	This typeface is use to indicate prototype code.
`Argument`	This typeface is used to indicate arguments.
`Name`	This typeface is used to indicate actual code or a programming construct.
register	This typeface is used to indicate a processor register.
[REF-xxx]	This is a reference to tagged reference document.

1.4 Glossary

The following table defines several terms that are used throughout this document.

Term	Description
VM	The Virtual Machine, a pseudo-processor implementation consisting of registers which are manipulated by the interpreter when executing EBC instructions.
Interpreter	The software implementation that decodes EBC binary instructions and executes them on a VM. Also called VM interpreter.
EBC	EFI byte code, which is the binary encoding of instructions as output by the EBC C compiler and linker. Executed by the interpreter.
Image	Executable binary file containing EBC and data. Output by the EBC linker.
Native code	Low level instructions that are native to the host processor. As such, the processor executes them directly with no overhead of interpretation. Contrast this with EBC, which must be interpreted by native code to operate on a VM.
COFF	Common Object File Format, a standard file format for binary images.
EBC image	Executable EBC image following the PE32+ file format.

2 Overview

The current design for option ROMs that are used in IA-32 personal computer systems has been in place since 1981. Attempts to change the basic design requirements have failed for a variety of reasons. This specification is attempting to help achieve the following goals:

1) Abstract and extensible design
2) CPU independence
3) OS independence
4) Build upon existing specifications when possible
5) Facilitate the removal of legacy infrastructure
6) Exclusive use of EFI Services

One way to satisfy many of these goals is to define a pseudo or virtual machine that can interpret a predefined instruction set. This will allow the virtual machine to be ported across CPU and system architectures without changing or recompiling the option ROM. This specification defines a set of machine level instructions that can be generated by a C compiler.

The following is a detailed description of the requirements placed on future option ROMs.

2.1 CPU Architecture Independence

Option ROM images shall be independent of IA-32 and IA-64 processor architectures. In order to abstract the architectural differences between CPUs (not just limited to IA-32 and IA-64) Option ROM images shall be portable byte streams. This model is presented below:

1) 64-bit C source code
2) The Pseudo Code image is the flashed image
3) The System BIOS implements the Pseudo Code interpreter
4) The interpreter handles 32 vs. 64 bit issues

Current Option ROM technology is CPU dependent and heavily reliant upon the existence of the PC-AT infrastructure. These dependencies inhibit the evolution of both hardware and software under the veil of “backward compatibility”. A solution that isolates the hardware and support infrastructure through abstraction will facilitate the uninhibited progression of technology.

2.2 OS Independent

Option ROMs shall not require or assume the existence of a particular OS.

2.3 EFI Compliant

Option ROMs compliance with EFI requires (but is not limited to) the following:

1) Little Endian layout
2) Single threaded model with interrupt polling if needed
3) Where EFI provides required services, EFI is used exclusively. These include:
    1. Console I/O
    2. Memory Management
    3. Timer services
    4. Global variable access
4) When an Option ROM provides EFI services, the EFI specification is strictly followed:
    1. Service/protocol installation
    2. Calling conventions
    3. Data structure layouts
    4. Guaranteed return on services

2.4 Coexistence of Legacy Option ROMs

The infrastructure shall support coexistent Legacy Option ROM and Portable Option ROM images. This case would occur, for example, when a PnP Card has both Legacy and Portable Option ROM images flashed. The details of the mechanism used to select which image to load is beyond the scope of this document. Basically, a legacy System BIOS would not recognize a Portable Option ROM and therefore would never load it. Conversely, an EFI Firmware Boot Manager would only load images that it supports.

The Portable Option ROM format must utilize a legacy format to the extent that a Legacy System BIOS can:

1) Determine the type of the image, in order to ignore the image. The type must be incompatible with currently defined types.
2) Determine the size of the image, in order to skip to the next image.

2.5 Image is Relocatable

An Option ROM image shall be eligible for placement in any system memory area large enough to accommodate it.

Current Option ROM technology requires images to be shadowed in system memory address range 0xC0000 to 0xEFFFF on a 2048 byte boundary. This dependency not only limits the number of Option ROMs, it results in unused memory fragments up to 2K bytes.

2.6 Size Restrictions Based on Memory Available

Option ROM images shall not be limited to a predetermined fixed maximum size (theoretically a big real-mode limit of 4 gigabytes applies).

Current Option ROM technology limits the size of a Pre-Initialization Option ROM image to 128K bytes (126K actual). Additionally, in the DDIM an image is not allowed to grow during initialization. It is inevitable that 64-bit solutions will increase in complexity and size. To avoid revisiting this issue, Portable Option ROM size is only limited by available system memory. EFI memory allocation services allow device drivers to claim as much memory as they need.

The PCI specification limits the size of an image stored in an option ROM to 16MB. If the driver is stored on the hard drive then the 16MB option ROM limit does not apply. Secondly, the PE/COFF object format limits this size of the image to 2GB.

3 The Virtual Machine

The virtual machine is composed of several parts: the operating environment, the instruction set, calls outside the virtual machine, and the load image.

3.1 Operating Environment

The virtual machine consists of 8 64-bit registers and an instruction execution engine. Most of the registers are general purpose in nature and can be used to store data or addresses. Some of the registers have specific definitions as documented below. Instructions may use the general form Rn to access the general-purpose registers, where n is a number between 0 and 7.

If an instruction requires an indirect access to memory, then the register is prefixed with @. For example **@R7** addresses the memory location pointed to by the address in **R7**.

If the register is used in an indexing operation, the register is suffixed with +/- offset. For example **@R6+0x10** addresses memory by adding 0x10 to the contents of **R6** and then using the result to access memory. Offsets can be 16, 32, or 64-bit signed values depending on the opcode.

Table 3-1 General Purpose VM Registers

Index	Register	Description
0	R0	Points to the top of the stack
1-3	R1-R3	Preserved across calls
4-7	R4-R7	Scratch, not preserved across calls

In addition to the general-purpose register, there are special purpose registers that may be accessed with certain instructions.

Table 3-2 Special VM Registers

Index	Register	Description
0	FLAGS	<table><tr><th>Bit</th><th>Description</th></tr><tr><td>0</td><td>Condition code</td></tr><tr><td>1..63</td><td>Reserved</td></tr></table>
1	IP	Points to current instruction
2..7	Reserved	Not defined

3.1.1 R0 Stack Pointer Register

The stack pointer is a 64-bit pointer that is acted on by CALL, RET, PUSH and POP instructions. The VM initializes this register to an area of free memory. This register can be used and modified like any other register. PUSH and CALL instructions decrement **R0** by 8 and then store their data. POP and RET instructions retrieve their data and then increment **R0** by 8.

3.1.2 Flags Special Register

The virtual machine defines one flag bit, the condition flag. This flag is set by the compare instructions and tested by the **JMP** instructions.

Table 3-3 VM FLAGS Register

Bit	Flag	Description
0	C	Set to 1 if the result of the last compare was true
1..63	Reserved	Reserved

3.1.3 IP Special Register

IP is the Instruction Pointer, which holds the address of the current instruction. The virtual machine will update **IP** to the address of the next instruction on completion of the current instruction, and will continue execution from the address indicated in **IP**. The **IP** register can be moved into a general-purpose register (**R1**-**R7**), then data manipulation and data movement instructions can manipulate the value and **FLAGS** can be set based on the results. This register may only be modified with the JMP/CALL and RET instructions. The instruction set is designed to use words as the minimum instruction entity. Therefore, the low order bit (bit 0) of **IP** is always cleared to 0. If a **RET** instruction would result in bit 0 being set to 1, then an exception occurs.

3.2 Instruction Encoding

‘Most instruction encodings follow the general form:

1 Byte Opcode | 1 Byte Operand | [Immediate data]

Table 3-4 Opcode Byte Encoding

Bit	Sym	Description
7	I	Immediate data present
6	W	Width
0-5	Op	The opcode of the instruction

If immediate data is present, the size of the immediate data is 16, 32, or 64 bits and is determined by the Opcode.

The width bit indicates the width of the operation, immediate data, or operand depending on the particular instruction.

For those instructions that use bit 6 of the opcode byte to indicate the size of the immediate data, if bit 7 of the opcode is 0 (no immediate data), then bit 6 is ignored by the VM.

3.3 Operand Register Encoding

Instructions are divided into categories based on the number of operands they require. Currently there are 0, 1, and 2-operand instructions. Operands can include immediate values, registers, and values accessed indirectly. Operands may also be a data source or destination. Operands take the following forms:

[@]Rn

One and two-operand instructions have an instruction byte followed by an operand byte. The instruction byte determines the number of operands and the size of the data that will be manipulated in each operand. Instructions that use an immediate value access the immediate data beyond the operand byte. Instructions have at most one immediate operand. The first operand is only indirect when the instruction opcode specifies an indirect instruction.

Table 3-5 Operand Byte Encoding

Bit	Description
7	0 = Operand 2 is direct 1 = Operand 2 is indirect
4..6	Operand 2 register
3	0 = Operand 1 is direct 1 = Operand 1 is indirect
0..2	Operand1 register

The location of an operand is determined by the operand's register, the operand's indirect bit, and if present and applicable to the operand the immediate data of the instruction. When an operand is indirect, immediate data may also be added to the specified address before the operand is loaded. This immediate data is called an index. The data load or store is always sized by the bit size of the operation as indicated by the instruction. On a load the value may be internally extended to perform the instruction.

Indexes can be 16, 32, or 64-bits wide based on instruction encoding.

Table 3-6 Index Encoding

Bit	Description
x+4	Sign bit
x+1..x+3	Bits assigned to natural units
A+1..x	Constant units (C)
0..a	Natural units (N)

The index is calculated according to the following equation: Index = C + N * (Size of pointer in bytes). The sign bit causes the index to be added or subtracted from the base register.

3.3.1 Sign Bit

The sign bit determines the sign of the index once the size calculation has been performed

3.3.2 Bits Assigned to Natural Units

This is a 3-bit field that determines the width of the natural units field. The units vary based on the size of the index according to the following table:

Table 3-7 Index Size in Index Encoding

Index Size	Units
16 bits	2 bits
32 bits	4 bits
64 bits	8 bits

3.3.3 Constant

The constant is the number of bytes in the index that do not scale with processor size. When the index is a 16-bit value, the maximum constant is 4095. This index is achieved when the bits assigned to natural units is 0.

3.3.4 Natural Units

Natural units are used when a structure has fields that can vary with the architecture of the CPU. Fields that break down into natural units include pointers and INTs. The size of one pointer equals one natural unit.

3.4 Immediate Operand Encoding

Immediate operands have two distinct methods of sizing. The first method is imposed by the instruction. Instructions can indicate that immediate values are 8, 16, 32, or 64 bits. All immediate values are two’s complement signed integers. Some immediate values are constants and others are offsets. The instruction encoding determines if constants are values or offsets. When an immediate operand is an offset the high order bit specifies if the value is a natural sized value, or is sized by the instruction. If the constant is natural sized the value is sign extended to the natural size of a pointer. It is possible to have a 64-bit constant on a 32-bit machine. If this happens an overflow is possible. This will result in a runtime fault.

4 Instruction Set

The VM instruction set is divided into 4 basic instruction types:

1) Program Flow
2) Compare
3) Data Manipulation
4) Data Movement

Some instructions do not follow this form. Differences are documented in the instruction definitions.

4.1 Program Flow Instructions

There are two types of program flow: those that change the **IP**, and those that terminate execution.

4.1.1 HALTing VM Execution

There is one instruction that may halt execution:

BREAK code

The BREAK instruction may exit the virtual machine. The code defines the type of BREAK and is limited to 0x100 codes. The BREAK instruction does not affect the FLAGS register.

Table 4-1 BREAK Instruction Encoding

<table>
<tr><th>Byte</th><th>Description</th></tr>
<tr><td>0</td><td>
<table>
<tr><th>Bit</th><th>Description</th></tr>
<tr><td>6..7</td><td>Reserved, must be 0</td></tr>
<tr><td>0..5</td><td>Opcode = 0x00 (BREAK)</td></tr>
</table>
</td></tr>
<tr><td>1</td><td>Break Code
<table>
<tr><th>Code</th><th>Description</th></tr>
<tr><td>0</td><td>Runaway program break</td></tr>
<tr><td>1</td><td>Virtual Machine Revision Number</td></tr>
<tr><td>2</td><td>Skip</td></tr>
<tr><td>3</td><td>Debug Break. The instruction is to break the flow of execution. No other information for the break is available. This encoding is commonly used for debugging.</td></tr>
<tr><td>4</td><td>System Call. This encoding is used to exit the virtual machine and invoke a system service.</td></tr>
<tr><td>5..255</td><td>Reserved</td></tr>
</table>
</td></tr>
</table>

BREAK 0 (Runaway Program Break)

The encoding for BREAK 0 is 0x00. If the VM attempts to execute a memory area that is zero it will result in a runaway break exception.

BREAK 1 (Virtual Machine Revision Number)

This function will return the revision of the VM in **R7**. The fields are in Binary Coded Decimal (BCD). A VM that conforms to this specification should return 0x1000 in **R7** when a BREAK 1 is executed.

Table 4-2 Virtual Machine Revision Number Encoding

Bits 17-63	Bits 12-16	Bits 8-11	Bits 4-7	Bits 0-3
Zero	Major	Minor	Sub-Minor	Least

BREAK 2 (Skip)

When this break is encountered, the next instruction is skipped. The VM will automatically advance the **IP** from the current instruction past the following instruction and allow execution to continue with the second instruction.

BREAK 3 (Debug Break)

Execution of a debug break instruction results in a debug break exception. If a debugger is attached or available, then it may halt execution of the image. Otherwise this instruction is ignored.

BREAK 4 (System Call)

System calls return information about the system, or perform an intrinsic system function. There are currently no system calls defined.

BREAK 5 (Create Thunk)

The BREAK 5 instruction causes the VM interpreter to create a thunk for the EBC entry point whose 64-bit address is stored at the location pointed to by VM register **R7.** The contents of the memory location pointed to by **R7** are then updated to point to the newly created thunk.

4.1.2 Instructions that change the IP

The JMP, CALL, and RET instructions change the **IP**.

4.1.2.1 JMP Instructions

JMP instructions have the form:

JMP[64][cs|cc] Imm64

JMP32[cs|cc] [@]R_i [Imm32]

JMP8[cs|cc] Imm8

There are many encodings of the JMP instruction. All encodings allow for conditional checks against the condition flag. If the conditional check fails, the JMP instruction does not alter the **IP** and execution continues at the next instruction. The condition flag is an output of the CMP instruction.

Table 4-3 JMP32/64 Instruction Encoding

<table>
<tr><th>Byte</th><th>Description</th></tr>
<tr><td>0</td><td>
<table>
<tr><th>Bit</th><th>Description</th></tr>
<tr><td>7</td><td>1 = Immediate data present</td></tr>
<tr><td>6</td><td>0 = 32 bit immediate data
1 = 64 bit immediate data</td></tr>
<tr><td>0..5</td><td>Opcode = 0x01 (JMP)</td></tr>
</table>
</td></tr>
<tr><td>1</td><td>JMP address operand
<table>
<tr><th>Bit</th><th>Description</th></tr>
<tr><td>7</td><td>0 = Unconditional
1 = Conditional</td></tr>
<tr><td>6</td><td>0 = CC (condition clear)
1 = CS (condition set)</td></tr>
<tr><td>5</td><td>Reserved</td></tr>
<tr><td>4</td><td>0 = Absolute address
1 = Relative address</td></tr>
<tr><td>3</td><td>0 = Operand1 direct
1 = Operand1 indirect</td></tr>
<tr><td>0..2</td><td>Operand1</td></tr>
</table>
</td></tr>
<tr><td>2..5</td><td>Optional 32 bit immediate data</td></tr>
<tr><td>2..9</td><td>Optional 64 bit immediate data</td></tr>
</table>

The JMP instruction uses bit 7 of the opcode byte to indicate if immediate data is present, and if so, bit 6 is used to indicate the width of the immediate data. This JMP opcode supports no immediate data, 32-bit immediate data, or 64-bit immediate data. There is a subsequent JMP opcode that supports 8-bit immediate data.

When bits 7 and 6 of the opcode byte are set to 1, the immediate data is 64 bits wide and the following criteria are true:

1) The index register is ignored
2) Bit 4 is cleared to 0 indicating an absolute address
3) Bit 3 is ignored

When bit 3 of the operand is set to 1 indicating that the instruction is an indirect jump, bit 4 of the operand shall be cleared to 0 indicating that the address is absolute.

Bit 7 of JMP addressing operand indicates if the JMP is conditional. If the jump is unconditional, then bit 6 of the operand byte is ignored. If the jump is conditional, bit 6 indicates whether the condition flag must be set or clear for the JMP to occur. If the JMP does not occur, execution continues at the instruction following the JMP. Bit 4 indicates if the JMP is relative to the current instructions IP or not. Operand1 indicates which 64-bit register to apply to the address calculation. And finally if the Operand1 indirect bit is set, the address is considered a 64-bit pointer to the actual 64-bit address.

When bits 0..3 of the operand are set to 0, the jump is an immediate PC-relative jump. In this case, the register is ignored and the immediate data is a signed offset of the appropriate size or an absolute address based on the encoding of bit 4 of the operand byte, sign extended to 64-bit value and added to the PC to obtain the target address.

The address calculation of a JMP32/64 occurs as:

```
JMP32/JMP64: (condition met)
Address = Operand1
If (Operand.Relative) {
      Address += IP of current instruction
}
If (Opcode.Immedate Data) {
      If (Opcode.32 Bit Data) {
            Immed = 32 bit value from Istream
            Address += 64 bit sign-extend Immed
      } else {
            Immed = 64 bit value from Istream
            Address += Immed
      }
}
if (Operand1.Indirect) {
      Address = 64 bit read of Address
}
Address is target JMP address
```

Table 4-4 JMP8 Instruction Encoding

Byte	Description
0	<table><tr><th>Bit</th><th>Description</th></tr><tr><td>7</td><td>0 = Unconditional 1 = Conditional</td></tr><tr><td>6</td><td>0 = CC (condition clear) 1 = CS (condition set)</td></tr><tr><td>0..5</td><td>Opcode = 0x02 (JMP8)</td></tr></table>
1	8 bit offset

The JMP8 instruction does not use any operands. Instead the second byte of the instruction is a signed 8-bit offset. All JMP8 encodings are relative jumps. The offset is in words. The offset shall be shifted left one bit before it is added to the **IP**. Bit 7 of the opcode indicates when the JMP8 is conditional. If so, bit 6 indicates whether the condition flag must be clear or set for the jump to occur. If the condition is not met, execution continues at the next instruction.

```
JMP8: (condition met)
Address = IP of current instruction
Immed = 8 bit value from Istream
Immed = 64 bit sign-extend Immed * 2
IP = Address + Immed
```

4.1.2.2 CALL Instructions

CALL instructions have the form:

CALL[32] [@]R_1 [Imm32]

CALLEX[32] [@]R_1 [Imm32]

CALL64 [@]R_1 [Imm64]

CALLEX64 [@] R_1 [Imm64]

CALL is a form of the jump instruction that pushes the address of the next instruction on the stack. When the program is to resume execution, the RET instruction pops the address from the stack into the **IP**. The call instruction pushes a 64-bit return address. The CALL instruction is not conditional.

Table 4-5 CALL Instruction Encoding

<table>
<tr><th>Byte</th><th>Description</th></tr>
<tr><td>0</td><td>
<table>
<tr><th>Bit</th><th>Description</th></tr>
<tr><td>7</td><td>1 = Immediate data present</td></tr>
<tr><td>6</td><td>0 = 32 bit immediate data
1 = 64 bit immediate data</td></tr>
<tr><td>0..5</td><td>Opcode = 0x03 (CALL)</td></tr>
</table>
</td></tr>
<tr><td>1</td><td>CALL/CALLEX address operand
<table>
<tr><th>Bit</th><th>Description</th></tr>
<tr><td>6..7</td><td>Reserved, must be 0</td></tr>
<tr><td>5</td><td>0 = Call to VM code
1 = Call to Native code</td></tr>
<tr><td>4</td><td>0 = Absolute address
1 = Relative address</td></tr>
<tr><td>3</td><td>0 = Operand1 direct
1 = Operand1 indirect</td></tr>
<tr><td>0..2</td><td>Operand1</td></tr>
</table>
</td></tr>
<tr><td>2..5</td><td>Optional 32 bit immediate data</td></tr>
<tr><td>2..9</td><td>Optional 64 bit immediate data</td></tr>
</table>

The CALL instruction uses bit 7 of the opcode byte to indicate if immediate data is present, and bit 6 to indicate the width of the immediate data. The CALL instruction supports: no immediate data, 32 bit immediate and 64 bit immediate data.

When bits 7 and 6 of the opcode byte are set to 1 the call is immediate, the immediate data is 64 bits wide and the following criteria are true:

1) The index register is ignored
2) Bit 4 is cleared to 0 indicating an absolute address
3) Bit 3 is ignored

When bit 3 of the operand is set to 1 indicating that the instruction is an indirect call, bit 4 of the operand shall be cleared to 0 indicating that the address is absolute.

The address operand is similar to the JMP operand except the condition code is never tested. Operand1 indicates which 64 bit register applies to the address calculation. If the operand1 indirect

bit is set, the address is considered a 64-bit pointer to the actual 64-bit address. When the indirect bit is set (bit 3) and immediate data is present, the immediate data is treated as an index.

When bits 0..3 of the operand are set to 0, the call is an immediate call. In this case, the register is ignored and the immediate data is a signed offset of the appropriate size or an absolute address based on the encoding of bit 4.

When bit 5 of the operand is cleared to 0, the destination address contains instructions that are interpreted by the virtual machine. When bit 5 of the operand is set to 1, the destination address contains native instructions. The address calculation of a CALL32/64 occurs as:

```
CALL32/CALL64:
Address = Operand1
If (Opcode.Immedate Data) {
      If (Opcode.32 Bit Data) {
            Immed = 32 bit value from Istream
            Address += 64 bit sign-extend Immed
      } else {
            Immed = 64 bit value from Istream
            Address += Immed
      }
}
if (Operand1.Indirect) {
      Address = 64 bit read of Address
}
Address is target CALL address
```

4.1.2.3 Return Instruction

The RET instruction has the form:

RET

The RET instruction moves 8 bytes of data from the stack to the **IP** and then adds 8 to **R0**.

Table 4-6 RET Instruction Encoding

Byte	Description
0	<table><tr><th>Bit</th><th>Description</th></tr><tr><td>6..7</td><td>Reserved, must be 0</td></tr><tr><td>0..5</td><td>Opcode = 0x04 (RET)</td></tr></table>
1	Reserved, must be 0

4.2 Compare Instructions

The compare instruction compares two operands and sets the result flag if the condition is true.

CMP[n]cc R_1, [@]R_2 [Imm16]

CMPI[d|q]cc R_1, [Imm16|32]

The following table describes the encoding for compare instructions. In the opcode column the first number is for CMP and the second number is for CMPI.

Table 4-7 CMP/CMPI Instruction Opcodes

Opcode CMP / CMPI	Cc	Description
0x05/0x2D	eq	Compare Signed Equal / Not Equal
0x06/0x2E	lte	Compare Signed Less Than or Equal / Greater Than
0x07/0x2F	gte	Compare Signed Greater Than or Equal / Less Than
0x08/0x30	ulte	Compare Unsigned Less Than or Equal / Greater Than
0x09/0x31	ugte	Compare Unsigned Greater Than or Equal / Less Than

Table 4-8 CMP Instruction Encoding

Byte	Description
0	<table><tr><th>Bit</th><th>Description</th></tr><tr><td>7</td><td>0=No immediate data 1=Immediate data present</td></tr><tr><td>6</td><td>0=32 bit operand width 1=64 bit operand width</td></tr><tr><td>0..5</td><td>CMP Opcodes</td></tr></table>
1	Operand
2..3	Optional 16 bit immediate data

Table 4-9 CMPI Instruction Encoding

Byte	Description
0	<table><tr><th>Bit</th><th>Description</th></tr><tr><td>7</td><td>0=16 bit immediate data 1=32 bit immediate data</td></tr><tr><td>6</td><td>0=32 bit compare 1=64 bit compare</td></tr><tr><td>0..5</td><td>CMPI Opcodes</td></tr></table>
1	Operand
2..3	16 bit immediate data
2..5	32 bit immediate data

If bit 6 of the opcode is set for CMP instructions, a 64-bit comparison is performed; otherwise, a 32-bit comparison is performed and the upper 32 bits are ignored.

If the instruction is CMPI, the width of the test is determined by bit 6 and the width of the immediate data is determined by bit 7.

The result of the CMP instruction is to set the condition flag based on the result of subtracting $\mathbf{R}_2$ from $\mathbf{R}_1$.

4.3 Data Manipulation Instructions

All the data manipulation instructions have two operands and use the following format:

OPCODE[32|64] [@]R_1,[@]R_2 [Imm16]

If bit 7 of the opcode is set, it indicates that 16 bits of immediate data is present, is sign extended and applied to register operand 2. If the indirect bit of operand 2 is set, the immediate value is then considered an index. Data is fetched based on bit 6 of the opcode as either a 32-bit quantity or 64-bit quantity.

If bit 6 of the opcode is not set, the math operation is performed as a 32-bit operation with the upper 32 bits being ignored. If the indirect bit of Operand1 is set, the resulting 32 bit value is stored at the 64 bit location pointed to by $\mathbf{R}_1$. If the indirect bit is not set, the 32-bit value is stored into $\mathbf{R}_1$ with the upper 32 bits being zero.

If bit 6 of the opcode is set, the math operation is perform as a full 64-bit operation and stored into $\mathbf{R}_1$ or the address pointed to by $\mathbf{R}_1$ as indicated by the indirect bit of Operand1.

Table 4-10 Data Manipulation Instruction Opcodes

Opcode	Description	Operation
0x0A	NOT[64|32] R_1,[@]R_2[Imm16]	R_1 = NOT R_2
0x0B	NEG[64|32] R_1,[@]R_2[Imm16]	R_1 = NEG R_2
0x0C	ADD[64|32] R_1,[@]R_2[Imm16]	R_1 = R_1+R_2
0x0D	SUB[64|32] R_1,[@]R_2[Imm16]	R_1 = R_1-R_2
0x0E	MUL[64|32] R_1,[@]R_2[Imm16]	R_1 = R_1*R_2
0x0F	MULU[64|32] R_1,[@]R_2[Imm16]	R_1 = R_1*R_2
0x10	DIV[64|32] R_1,[@]R_2[Imm16]	R_1 = R_1/R_2 (Integer Divide)
0x11	DIVU[64|32] R_1,[@]R_2[Imm16]	R_1 = R_1/R_2 (Integer Divide)
0x12	MOD[64|32] R_1,[@]R_2[Imm16]	R_1 = R_1 mod R_2
0x13	MODU[64|32] R_1,[@]R_2[Imm16]	R_1 = R_1 mod R_2
0x14	AND[64|32] R_1,[@]R_2[Imm16]	R_1 = R_1 and R_2
0x15	OR[64|32] R_1,[@]R_2[Imm16]	R_1 = R_1 or R_2
0x16	XOR[64|32] R_1,[@]R_2[Imm16]	R_1 = R_1 xor R_2
0x17	SHL[64|32] R_1,[@]R_2[Imm16]	R_1 = R_1 shl R_2
0x18	SHR[64|32] R_1,[@]R_2[Imm16]	R_1 = R_1 shr R_2
0x19	ASHR[64|32] R_1,[@]R_2[Imm16]	R_1 = R_1 ashr R_2
0x1A	EXTNDB [@]R_1,[@]R_2[Imm16]	Extract Byte R_2, sign extend to 64 bits and store back in R_1
0x1B	EXTNDW| [@]R_1,[@]R_2[Imm16]	Extract Word from R_2, sign extend to 64 bits and store back in R_1
0x1C	EXTNDD [@]R_1,[@]R_2[Imm16]	Extract Dword from R_2, sign extend to 64 bits and store back in R_1

Table 4-11 Data Manipulation Instruction Encodings

Byte	Description
0	<table><tr><th>Bit</th><th>Description</th></tr><tr><td>7</td><td>0 = No immediate data 1 = Immediate data present</td></tr><tr><td>6</td><td>0 = 32 bit width operation 1 = 64 bit width operation</td></tr><tr><td>0..5</td><td>Data Manipulation Opcodes</td></tr></table>
1	Operand
2..3	Optional 16 bit immediate data

4.3.1 Fault Handling

If the DIV or MOD instruction results in a divide by 0, then a divide by 0 exception occurs.

4.4 Data Movement Instructions

The data movement instructions move data between the registers and memory. There are two forms: 1-operand and 2-operand instructions. The basic MOV instruction is a 2-operand move. It has both a source and a result operand. PUSH and POP instructions take only one explicit operand.

4.4.1 2-Operand Moves

Data is moved from $\mathbf{R}_2$ to $\mathbf{R}_1$. Only the numbers of bits specified by the data type are moved from the source and stored in the result. The remaining high order bits are zeroed when the destination is a register.

The source or resulting addresses do not need to be naturally aligned.

4.4.1.1 Move From Indexed Operand

This type of move takes the form:

MOV[b|w|d|q|n][w|d|q] [@]R_1,[@]R_2 [Imm16|32|64]

Move from indexed operand moves data from and effective address that may include an index on the source operand. The destination may also be indirect but in cannot be indexed. The first optional character specifies the width of the move and can be natural (n), 8 (b), 16 (w), 32 (d), or 64 (q) bits. The default is 64 bits. The second optional character specifies the size of the constant or index, which can be 16 (w), 32 (d), or 64 (q) bits.

4.4.1.2 Move To Index Operand

This type of move takes the form:

MOV[b|w|d|q|n][w|d|q] [@]R_1[Imm16|32|64], [@]R_2

Move to indexed operand moves data from an effective address that may be indirect to a destination that may be both indirect and indexed. The source may also be indirect but it cannot be indexed. The first character specifies the width of the move and can be natural (n), 8 (b), 16 (w), 32 (d), or 64 (q) bits. The default is 64 bits. The second character specifies the size of the constant or index, which can be 16 (w), 32 (d), or 64 (q) bits.

Table 4-12 MOV Opcodes

Opcode	Description
0x1D	MOVbw [@]R_1,[@]R_2[Imm16] MOVbw [@]R_1[Imm16],[@]R_2
0x1E	MOVww [@]R_1,[@]R_2[Imm16] MOVww [@]R_1[Imm16],[@]R_2
0x1F	MOVdw [@]R_1,[@]R_2[Imm16] MOVdw [@]R_1[Imm16],[@]R_2
0x20	MOVqw [@]R_1,[@]R_2[Imm16] MOVqw [@]R_1[Imm16],[@]R_2
0x21	MOVbd [@]R_1,[@]R_2[Imm32] MOVbd [@]R_1[Imm32],[@]R_2
0x22	MOVwd [@]R_1,[@]R_2[Imm32] MOVwd [@]R_1[Imm32],[@]R_2
0x23	MOVdd [@]R_1,[@]R_2[Imm32] MOVdd [@]R_1[Imm32],[@]R_2
0x24	MOVqd [@]R_1,[@]R_2[Imm32] MOVqd [@]R_1[Imm32],[@]R_2
0x25	MOVbq [@]R_1,[@]R_2[Imm64] MOVbq [@]R_1[Imm64],[@]R_2
0x26	MOVwq [@]R_1,[@]R_2[Imm64] MOVwq [@]R_1[Imm64],[@]R_2
0x27	MOVdq [@]R_1,[@]R_2[Imm64] MOVdq [@]R_1[Imm64],[@]R_2
0x28	MOVqq [@]R_1,[@]R_2[Imm64] MOVqq [@]R_1[Imm64],[@]R_2
0x32	MOVnw [@]R_1,[@]R_2[Imm16] MOVnw [@]R_1[Imm16],[@]R_2
0x33	MOVnd [@]R_1,[@]R_2[Imm32] MOVnd [@]R_1[Imm32],[@]R_2
0x34	MOVnq [@]R_1,[@]R_2[Imm64] MOVnq [@]R_1[Imm64],[@]R_2

4.4.1.3 Move Immediate To Index Operand

This type of move takes the form:

MOVI[w|d|q][b|w|d|q] [@]R_1[Imm16], Imm16|32|64

MOVIn[w|d|q] [@]R_1[Imm16], Imm16|32|64

Move to indexed operand moves immediate data to a destination that may be both indirect and indexed. The optional characters specify the width of the immediate data and the width of the move. MOVI and MOVIn (natural move immediate) immediate data can be 16 (w), 32 (d), or 64 (q) bits as specified by the first character. The actual width of the MOVI can be 8 (b), 16 (w), 32 (d) or 64 (q) bits.

Table 4-13 MOV Instruction Encoding

<table>
<tr><th>Byte</th><th>Description</th></tr>
<tr><td>0</td><td>
<table>
<tr><th>Bit</th><th>Description</th></tr>
<tr><td>7</td><td>0 = No immediate data
1 = Immediate data present</td></tr>
<tr><td>6</td><td>0 = Immediate data applies to R_1
1 = Immediate data applies to R_2</td></tr>
<tr><td>0..5</td><td>MOV Opcodes</td></tr>
</table>
</td></tr>
<tr><td>1</td><td>Operand</td></tr>
<tr><td>2..3</td><td>Optional 16 bit immediate data</td></tr>
<tr><td>2..5</td><td>Optional 32 bit immediate data</td></tr>
<tr><td>2..9</td><td>Optional 64 bit immediate data</td></tr>
</table>

Table 4-14 MOVI Instruction Encoding

<table>
<tr><th>Byte</th><th>Description</th></tr>
<tr><td>0</td><td>
<table>
<tr><th>Bit</th><th>Description</th></tr>
<tr><td>6..7</td><td>0 = Reserved
1 = Data is a 16 bits
2 = Data is 32 bits
3 = Data is 64 bits</td></tr>
<tr><td>0..5</td><td>Opcode = 0x37 (MOVI)
Opcode = 0x38 (MOVIn)</td></tr>
</table>
</td></tr>
<tr><td>1</td><td>
<table>
<tr><th>Bit</th><th>Description</th></tr>
<tr><td>7</td><td>Reserved</td></tr>
<tr><td>6</td><td>0 = Optional immediate absent
1 = Optional immediate present</td></tr>
<tr><td>4..5</td><td>0 = 8 (b) width of move
1 = 16(w) width of move
2 = 32(w) width of move
3 = 64(q) width of move</td></tr>
<tr><td>3</td><td>0 = Operand 1 direct
1 = Operand 1 indirect</td></tr>
<tr><td>0..2</td><td>Operand 1</td></tr>
</table>
</td></tr>
<tr><td>2..3</td><td>16 bit immediate index (Optional)</td></tr>
<tr><td>4..5</td><td>16 bit immediate data</td></tr>
<tr><td>4..7</td><td>32 bit immediate data</td></tr>
<tr><td>4..11</td><td>64 bit immediate data</td></tr>
</table>

4.4.1.4 Special Register 2-Operand Moves

The special register instructions can move values between a special register and a general-purpose register. These instructions do not support indirection and the size of the moved data is always 64 bits. LOAD places data into the special purpose register. STORE moves data from the special purpose register to the general register.

If you try to load the **IP** the instruction becomes a NOP. If the debugger is on, a fault is generated

LOADSP SP_1,R_2

STORESP R_1, SP_2

Table 4-15 LOADSP Instruction Encoding

Byte	Description
0	<table><tr><th>Bit</th><th>Description</th></tr><tr><td>6..7</td><td>Reserved, must be 0</td></tr><tr><td>0..5</td><td>Opcode = 0x29 (LOADSP)</td></tr></table>
1	<table><tr><th>Bit</th><th>Description</th></tr><tr><td>7</td><td>Reserved, must be 0</td></tr><tr><td>4..6</td><td>Operand2. General purpose register</td></tr><tr><td>3</td><td>Reserved, must be 0</td></tr><tr><td>0..2</td><td>Operand1. Special purpose register.</td></tr></table>

Table 4-16 STORESP Instruction Encoding

Byte	Description
0	<table><tr><th>Bit</th><th>Description</th></tr><tr><td>6..7</td><td>Reserved, must be 0</td></tr><tr><td>0..5</td><td>Opcode = 0x2A (STORESP)</td></tr></table>
1	<table><tr><th>Bit</th><th>Description</th></tr><tr><td>7</td><td>Reserved, must be 0</td></tr><tr><td>4..6</td><td>Operand1. Special purpose register.</td></tr><tr><td>3</td><td>Reserved, must be 0</td></tr><tr><td>0..2</td><td>Operand2. General-purpose register.</td></tr></table>

4.4.2 1-Operand Moves

The PUSH and POP instructions require only one operand. These instructions move data to and from the stack and maintain the stack pointer.

Table 4-17 PUSH/POP Opcodes

Opcode	Description
0x2B	PUSH [@]R_1 [Imm16]
0x2C	POP [@]R_1 [Imm16]
0x35	PUSHn [@]R_1 [Imm16]
0x36	POPn [@]R_1 [Imm16]

Table 4-18 PUSH/POP Instruction Encoding

Byte	Description
0	<table><tr><th>Bit</th><th>Description</th></tr><tr><td>7</td><td>1=Immediate data present 0=No immediate data</td></tr><tr><td>6</td><td>1=64 bit operation 0=32 bit operation</td></tr><tr><td>0..5</td><td>PUSH or POP Opcode</td></tr></table>
1	<table><tr><th>Bit</th><th>Description</th></tr><tr><td>4..7</td><td>Reserved, must be 0</td></tr><tr><td>3</td><td>Operand1 indirect</td></tr><tr><td>0..2</td><td>Operand1 register</td></tr></table>
2..3	Optional 16 bit immediate data

5 Runtime & Software Conventions

5.1 Calling Outside VM

Calls can be made to routines in other modules that are native or in other VM. It is the responsibility of the calling VM to prepare the outgoing arguments correctly to the call outside the VM. It is also the responsibility of the VM to prepare the incoming arguments correctly for the call from outside the VM.

Calls outside VM must use the external CALLEX opcode.

5.2 Calling Inside VM

Calls inside VM can be made either directly using the CALL opcode or using the external CALLEX opcode instructions. Using direct CALL opcode is an optimization.

5.3 Parameter Passing

Parameters are pushed on the stack in order of right to left.

All parameters are stored or accessed as natural size (using naturally sized instruction) except 64bit integers, which are pushed as 64-bit values. 32-bit integers are pushed as natural size (since they should be passed as 64-bit parameter values on IA64).

5.4 Return Values

Return values less than 8 bytes are returned in register **R7**. Any return value greater then 8 bytes will be returned in memory with the caller allocating the memory space and passing the address as first argument.

5.5 Binary Format

PE32+ format will be used for generating binaries for the VM. A new section will be added to the binary format. The new section will be the VarBss section. All global and static variables will be placed in this section. The size of the section is dependent on the architecture (32 bit or 64 bit) on which the binary will be run. The size field in the section will have two fields: scaled size and constant size. To find the actual size of the section, multiply the scaled size by the pointer size defined by the architecture, and then add to constant size.

Variables containing pointers that are initialized will also be placed in the VarBss section, with the compiler generating code to initialize at runtime.

6 Architectural Requirements

This chapter provides a high level overview of the architectural requirements that are necessary to support execution of EBC on a platform.

6.1 EBC Image Requirements

All EBC images will be PE32+ format. Some minor additions to the format will be required to support EBC images. See [REF-PECOFF] for details of this image file format.

An EBC image must be able to be executed unchanged on different platforms, independent of 32- or 64-bit processor. All EBC images should be driver implementations.

6.2 EBC Execution Interfacing Requirements

EBC drivers will typically be designed to execute in an (usually pre-boot) EFI environment. As such, EBC drivers must be able to invoke protocols, and expose protocols for use by other drivers or applications. The following execution transitions must be supported:

- EBC calling EBC
- EBC calling native code
- Native code calling EBC
- Native code calling native code
- Returning from all the above transitions

Obviously native code calling native code is available by default, so is not discussed in this document.

To maintain backward compatibility with existing native code, and minimize the overhead for non-EBC drivers calling EBC protocols, all four transitions must be seamless from the application perspective. Therefore, drivers, whether EBC or native, shall not be required to have any knowledge of whether or not the calling code, or the code being called, is native or EBC compiled code. The onus is put on the tools and VM interpreter to support this requirement.

6.3 Interfacing Function Parameters Requirements

To allow code execution across protocol boundaries, the VM interpreter must ensure that parameters passed across execution transitions are handled in the same manner as the standard parameter passing convention for the native processor.

6.4 Function Returns Requirements

The VM interpreter must support standard function returns to resume execution to the caller of external protocols. The details of this requirement are specific to the native processor. The called function must not be required to have any knowledge of whether or not the caller is EBC or native code.

6.5 Function Return Values Requirements

The VM interpreter must support standard function return values from called protocols. The exact implementation of this functionality is dependent on the native processor. This requirement applies to return values of 64 bits or less. The called function must not be required to have any knowledge of whether or not the caller is EBC or native code. Note that returning of structures is not supported.

6.6 VM Interpreter Protocol

Summary

This protocol provides the services to allow execution of EBC images.

GUID

```
#define EFI_VM_PROTOCOL_GUID \
            {0x13AC6DD0L, 0x73D0, 0x11D4, 0xB0, 0x6B, \
            0x00, 0xAA, 0x00, 0xBD, 0x6D, 0xE7}
```

Revision Number

```
#define EFI_VM_PROTOCOL_REVISION  ((1<<16)| 10)
```

Protocol Interface Structure

```
typedef struct _EFI_VM_PROTOCOL {
  UINT64                    Revision;
  EFI_VM_CREATE_THUNK       CreateThunk;
  EFI_VM_UNLOAD_IMAGE       UnloadImage;
} EFI_VM_PROTOCOL;
```

Parameters

Revision	Revision of the **EFI_VM_PROTOCOL**. All future revisions will be backward compatible to the current revision.
CreateThunk	Creates a thunk for an EBC image entry point or protocol service, and returns a pointer to the thunk.

UnloadImage	Called when an EBC image is unloaded to allow the interpreter to perform any cleanup associated with the image's execution.

Description

The EFI VM protocol provides services to execute EBC images, which will typically be loaded into option ROMs. The image loader will load the EBC image, perform standard relocations, and invoke the CreateThunk service to create a thunk for the EBC image's entry point. The image can then be run using the standard EFI start image services.

6.6.1 EFI_VM_PROTOCOL.CreateThunk

Summary

Creates a thunk for an EBC entry point, returning the address of the thunk.

Prototype

```
typedef
EFI_STATUS
(* EFI_VM_CREATE_THUNK) (
  IN EFI_HANDLE       ImageHandle,
  IN VOID             *EbcEntryPoint,
  OUT  VOID           **Thunk
  );
```

Parameters

ImageHandle	Handle of image for which the thunk is being created.
EbcEntryPoint	Address of the actual EBC entry point or protocol service the thunk should call.
Thunk	Returned pointer to a thunk created.

Description

A PE32+ EBC image, like any other PE32+ image, contains an optional header which defines the entry point for image execution. However for EBC images this is the entry point of EBC, so is not directly executable by the native processor. Therefore when an EBC image is loaded, the loader must call this service to get a pointer to native code (thunk) that can be executed which will invoke the VM interpreter to begin execution at the original EBC entry point.

Status Codes Returned

EFI_SUCCESS	The function completed successfully.
EFI_INVALID_PARAMETER	Image entry point is not 2-byte aligned.
EFI_OUT_OF_RESOURCES	Memory could not be allocated for the thunk

6.6.2 EFI_VM_PROTOCOL.UnloadImage

Summary

Called prior to unloading an EBC image from memory.

Prototype

```
typedef
EFI_STATUS
(* EFI_VM_UNLOAD_IMAGE) (
  IN EFI_HANDLE  ImageHandle
  );
```

Parameters

ImageHandle Image handle of the EBC image that is being unloaded from memory.

Description

This function is called after an EBC image has exited, but before the image is actually unloaded. It is intended to provide the interpreter with the opportunity to perform any cleanup that may be necessary as a result of executing the image.

Status Codes Returned

EFI_SUCCESS	The function completed successfully.
EFI_INVALID_PARAMETER	Image handle is not recognized as belonging to an EBC image that has been executed.

7 EBC Tools

7.1 EBC C Compiler

This section describes the responsibilities of the EBC C compiler. To fully specify these responsibilities requires that the thunking mechanisms between EBC and native code be described.

7.1.1 C Coding Convention

The EBC C compiler supports only the C programming language. There is no support of inline assembly, floating point types/operations, and C calling conventions other than CDECL.

Pointer type in C is supported only as 64-bit pointer. The code should be 64-bit pointer ready (not assign pointers to integers and vice versa).

The compiler does not support user-defined sections through pragmas

Global variables containing pointers that are initialized will be put in the uninitialized VarBss section and the compiler will generate code to initialize these variables during load time. The code will be placed in an init text section. This compiler-generated code will be executed before the actual image entry point is executed.

7.1.2 EBC Interface Assembly Instructions

The EBC instruction set includes two forms of a CALL instruction that can be used to invoke external protocols. Their assembly language formats are:

CALLEX Imm64

CALLEX32 [@]R_1 [Imm32]

Both forms can be used to invoke external protocols at an absolute address specified by the immediate data and/or register operand. The second form also supports jumping to code at a relative address. When one of these instructions is executed, the VM interpreter is responsible for thunking arguments and then jumping to the destination address. When the called function returns, code begins execution at the instruction following the CALL instruction. The process by which this happens is called thunking. Later sections describe this operation in detail.

7.1.3 Stack Maintenance and Argument Passing

There are several EBC assembly instructions that directly manipulate the stack contents and stack pointer. These instructions operate on the EBC stack, not the VM interpreter stack. The instructions include the following:

PUSHn [@]R_1 [Imm16]

POPn [@]R_1 [Imm16]

These instructions must adjust the EBC stack pointer in the same manner as equivalent instructions of the native instruction set. With this implementation, parameters pushed on the stack by an EBC driver can be accessed normally for stack-based native code. If native code expects parameters in registers, then the interpreter thunking process must transfer the arguments from EBC stack to the appropriate processor registers. The process would need to be reversed when native code calls EBC.

7.1.4 Native to EBC Arguments Calling Convention

The calling convention for arguments passed to EBC functions follows the standard calling convention. The arguments must be pushed as their native size. After the function arguments have been pushed on the stack, execution is passed to the called EBC function. The overhead of thunking the function parameters depends on the standard parameter passing convention for the host processor. The implementation of this functionality is left to the interpreter.

7.1.5 EBC to Native Arguments Calling Convention

When EBC makes function calls via function pointers, the EBC C compiler cannot determine whether the calls are to native code or EBC. It therefore assumes that the calls are to native code, and emits the appropriate EBC CALLEX instructions. To be compatible with calls to native code, the calling convention of EBC calling native code must follow the parameter passing convention. The EBC C compiler generates EBC that pushes all arguments on the stack. The VM interpreter is then responsible for performing the necessary thunking. The exact implementation of this functionality is left to the interpreter.

7.1.6 EBC to EBC Arguments Calling Convention

If the EBC C compiler is able to determine that a function call is to a local function, it can emit a standard EBC CALL instruction. In this case, the function arguments are passed as described in [REF-EFI].

7.1.7 Function Returns

When EBC calls an external function, the thunking process includes setting up the host processor stack or registers such that when the called function returns, execution is passed back to the EBC at the instruction following the call. The implementation is left to the interpreter, but it must follow the standard function return process of the host processor. Typically this will require the interpreter to push the return address on the stack or move it to a processor register prior to calling the external function.

7.1.8 Function Return Values

EBC function return values of 8 bytes or less are returned in VM register **R7**. Returning values larger than 8 bytes is not supported. If an EBC function returns to native code, then the interpreter thunking process is responsible for transferring the contents of **R7** to an appropriate location such that the caller has access to the value using standard native code. Typically the value will be transferred to a processor register. Conversely, if a native function returns to an EBC function, the interpreter is responsible for transferring the return value from the native return memory or register location into VM register **R7**.

7.1.9 Thunking

Thunking is the process by which transitions between execution of native and EBC are handled. The major issues that must be addressed for thunking are the handling of function arguments, how the external function is invoked, and how return values and function returns are handled. The following sections describe the thunking process for the possible transitions.

7.1.9.1 Thunking EBC to Native Code

By definition, all external calls from within EBC are calls to native code. The EBC CALLEX instructions are used to make these calls. A typical application for EBC calling native code would be a simple “Hello World” driver. For an EFI driver, the code could be written as shown below.

```
EFI_STATUS main(
    IN EFI_HANDLE           ImageHandle,
    IN EFI_SYSTEM_TABLE     *ST
    )
{
   ST->ConOut->OutputString(ST->ConOut, L”Hello World!”);
   return EFI_SUCCESS;
}
```

This C code, when compiled to EBC assembly, would result in two PUSHN instructions to push the parameters on the stack, some code to get the absolute address of the OutputString() function, then a CALLEX instruction to jump to native code. Typical pseudo assembly code for the function call would be something like the following:

```
PUSHn       _HelloString
PUSHn       _ConOut
MOVq        R1, _OutputString
CALLEX64    R1
```

The interpreter is responsible for executing the PUSHn instructions to push the arguments on the EBC stack when interpreting the PUSHn instructions. When the CALLEX instruction is encountered, it must thunk to external native code. The exact thunking mechanism is native

processor dependent. For example, an IA32 thunking implementation could simply move the system stack pointer to point to the EBC stack, then perform a CALL to the absolute address specified in VM register **R1**. However the IPF function calling convention calls for the first 8 function arguments being passed in registers. Therefore the IPF thunking mechanism requires the arguments to be copied from the EBC stack into processor registers. Then a CALL can be performed to jump to the absolute address in VM register **R1**. Note that since the interpreter is not aware of the number of arguments to the function being called, the maximum amount of data may be copied from the EBC stack into processor registers.

7.1.9.2 Thunking Native Code to EBC

An EBC driver may install protocols for use by other EBC drivers or EFI drivers or applications. These protocols provide the mechanism by which external native code can call EBC. Typical EFI C code to install a generic protocol is shown below.

```
EFI_STATUS Foo(UINT32 Arg1, UINT32 Arg2);

MyProtInterface->Service1= Foo;

Status = LibInstallProtocolInterfaces (&Handle, &MyProtGUID,
MyProtInterface, NULL);
```

To support thunking native code to EBC, the EBC compiler resolves all EBC function pointers using one level of indirection, similar to the plabel mechanism in IPF code. In this way, the address of an EBC function actually becomes the address of a piece of native (thunk) code that invokes the interpreter to execute the actual EBC function. As a result of this implementation, any time the address of an EBC function is taken, the EBC C compiler must generate the following:

- A 64-bit plabel data object that contains the actual address of the EBC function
- EBC initialization code that is executed before the image entry point that will execute EBC BREAK 5 instructions to create thunks for each plabel data object
- Associated relocations for the above

So for the above code sample, the compiler must generate EBC initialization code similar to the following. This code is executed prior to execution of the actual EBC driver's entry point.

```
MOVqq  R7, Foo_plabel    ; get address of Foo plabel
BREAK  5                 ; create a thunk for the function
```

The BREAK instruction causes the interpreter to create native thunk code elsewhere in memory, and then modify the memory location pointed to by R7 to point to the newly created thunk code for EBC function Foo. From within EBC, when the address of Foo is taken, the address of the thunk is actually returned. So for the assignment of the protocol Service1 above, the EBC C compiler will generate something like the following:

```
MOVqq  R7, Foo_plabel    ; get address of Foo plabel
MOVqq  R7, @R7           ; one level of indirection
MOVn   R6, _MyProtInterface->Service1 ; get address of variable
MOVqq  @R6, R7           ; address of thunk to ->Service1
```

7.1.9.3 Thunking EBC to EBC

EBC can call EBC via function pointers or protocols. These two mechanisms are treated identically by the EBC C compiler, and are performed using EBC CALLEX instructions. For EBC to call EBC, the EBC being called must have provided the address of the function. As described above, the address is actually the address of native thunk code for the actual EBC function. Therefore, when EBC calls EBC, the interpreter assumes native code is being called so prepares function arguments accordingly, and then makes the call. The native thunk code assumes native code is calling EBC, so will basically “undo” the preparation of function arguments, and then invoke the interpreter to execute the actual EBC function of interest.

7.2 EBC Linker

New constants must be defined for use by the linker in processing EBC images. The linker must also recognize the EBC image type IMAGE_FILE_MACHINE_EBC = 0x0EBC.

Otherwise only minimal changes will be required to the linker to support EBC images.

7.3 Image Loader

The EFI image loader is responsible for loading an executable image into memory and performing any fixups prior to execution of the image. For EBC images, the image loader must also invoke the interpreter protocol to create a thunk for the image entry point and return the address of this thunk. After loading the image in this manner, the image can be executed in the standard manner. To implement this functionality, only minor changes will be made to EFI service LoadImage(), and no changes should be made to StartImage().

7.4 Debug Support

The VM interpreter must support debugging in an EFI environment per [REF-DEBUG].

8 VM Exception Handling

This section lists the different types of exceptions that the VM may encounter during execution of an EBC image. If a debugger is attached to the EBC driver, then the debugger should be able to capture and identify the exception type. If a debugger is not attached, then the VM interpreter, upon encountering an exception, will halt execution of the current EBC image and return –1 to the caller.

8.1 Stack Overflow

Stack overflow can occur if the VM interpreter detects that function nesting within the interpreter or system interrupts was sufficient to potentially corrupt the EBC image's stack contents.

8.2 Stack Underflow

A stack underflow exception will occur if the EBC driver adjusts the stack pointer outside the range allocated to the driver.

8.3 Debug Break

A debug break exception occurs if the VM encounters a BREAK instruction with a break code of 3. If a debugger is not attached to the current EBC process then the VM interpreter will ignore this instruction.

8.4 Runaway Break

A runaway break exception occurs if the VM encounters a BREAK instruction with a break code of 0. This typically will occur if the VM attempts to execute EBC from cleared memory.

8.5 Invalid Opcode

An invalid opcode exception will occur if the VM interpreter encounters a reserved opcode during execution.

8.6 Opcode Encoding Error

An opcode encoding error exception can occur for the following:

- CALL instruction with the least significant bit of the destination address set
- A RET instruction with the least significant bit of the return address set

- LOADSP instructions that attempt to load the IP

8.7 Divide By 0

A divide-by-0 exception can occur for the EBC instructions DIV, DIVU, MOD, and MODU.

8.8 Unaligned Access

An unaligned access exception can occur if the particular implementation of the VM interpreter does not support unaligned accesses to data.

9 Option ROM Formats

The new option ROM capability is designed to be a departure from the legacy method of formatting an option ROM. PCI local bus add-in cards are the primary targets for this design although support for future bus types will be added as necessary. EFI Portable Byte Stream Drivers can be stored in option ROMs or on hard drives in an EFI system partition.

The new format defined for the EFI specification is intended to co-exist with legacy format PCI Expansion ROM images. This provides the ability for IHVs to make a single option ROM binary that contains both legacy and new format images at the same time. This is important for the ability to have single add-in card SKUs that can work in a variety of systems both with and without native support for EFI. Support for multiple image types in this way provides a smooth migration path during the period before widespread adoption of EFI drivers as the primary means of support for software needed to accomplish add-in card operation in the pre-OS boot timeframe.

9.1 EFI Drivers for PCI Add-in Cards

The location mechanism for EFI drivers in PCI option ROM containers is described fully in chapter 18 of the EFI 1.0 specification. Readers should refer to that specification for complete details of the scheme and associated data structures.

A full description of how to embed an EFI driver in an option ROM can be found in the EFI 1.1 PCI driver model specification.

9.2 Non-PCI Bus Support

EFI expansion ROMs are not supported on any other bus besides PCI local bus in the current revision of the EFI specification.

This means that support for EFI drivers in legacy ISA add-in card ROMs is explicitly excluded.

Support for EFI drivers to be located on add-in card type devices for future bus designs other than PCI local bus will be added to future revisions of the EFI specification. This support will depend upon the specifications that govern such new bus designs with respect to the mechanisms defined for support of driver code on devices.

A
Opcode Summary

Table A-1 - Opcode Summary

Opcode	Description
0x00	BREAK
0x01	JMP[32|64]
0x02	JMP8
0x03	CALL[32|64]
0x04	RET
0x05	CMPeq
0x06	CMPlte
0x07	CMPgte
0x08	CMPulte
0x09	CMPugte
0x0A	NOT[64|32] R_1,[@]R_2[Imm16]
0x0B	NEG[64|32] R_1,[@]R_2[Imm16]
0x0C	ADD[64|32] R_1,[@]R_2[Imm16]
0x0D	SUB[64|32] R_1,[@]R_2[Imm16]
0x0E	MUL[64|32] R_1,[@]R_2[Imm16]
0x0F	MULU[64|32] R_1,[@]R_2[Imm16]
0x10	DIV[64|32] R_1,[@]R_2[Imm16]
0x11	DIVU[64|32] R_1,[@]R_2[Imm16]
0x12	MOD[64|32] R_1,[@]R_2[Imm16]
0x13	MODU[64|32] R_1,[@]R_2[Imm16]
0x14	AND[64|32] R_1,[@]R_2[Imm16]
0x15	OR[64|32] R_1,[@]R_2[Imm16]
0x16	XOR[64|32] R_1,[@]R_2[Imm16]
0x17	SHL[64|32] R_1,[@]R_2[Imm16]
0x18	SHR[64|32] R_1,[@]R_2[Imm16]
0x19	ASHR[64|32] R_1,[@]R_2[Imm16]
0x1A	EXTNDB [@]R_1,[@]R_2[Imm16]
0x1B	EXTNDW| [@]R_1,[@]R_2[Imm16]
0x1C	EXTNDD [@]R_1,[@]R_2[Imm16]
0x1D	MOVbw
0x1E	MOVww

Opcode	Description
0x1F	MOVdw
0x20	MOVqw
0x21	MOVbd
0x22	MOVwd
0x23	MOVdd
0x24	MOVqd
0x25	MOVbq
0x26	MOVwq
0x27	MOVdq
0x28	MOVqq
0x29	LOADSP SP_1,R_2
0x2A	STORESP R_1, SP_2
0x2B	PUSH R_1[Imm16]
0x2C	POP R_1[Imm16]
0x2D	CMPIeq
0x2E	CMPIlte
0x2F	CMPIgte
0x30	CMPIulte
0x31	CMPIugte
0x32	MOVnw
0x33	MOVnd
0x34	MOVnq
0x35	PUSHn R_1[Imm16]
0x36	POPn R_1[Imm16]
0x37	MOVI
0x38	MOVIn
0x39	Reserved
0x3A	Reserved
0x3B	Reserved
0x3C	Reserved
0x3D	Reserved
0x3E	Reserved
0x3F	Reserved